Decltype and auto

Programming Language C++

Document no: N1478=03-0061

Jaakko Järvi
Indiana University
Pervasive Technology Laboratories
Bloomington, IN
jajarvi@osl.iu.edu

Bjarne Stroustrup
ATT&T Research
and Texas A&M University
bs@research.att.com

Douglas Gregor
Rensselaer Polytechnic Institute
Computer Science
Troy, NY
gregod@cs.rpi.edu

Jeremy Siek
Indiana University
Pervasive Technology Laboratories
Bloomington, IN
jsiek@osl.iu.edu

April 28, 2003

1 Introduction

C++ does not have a mechanism for querying the type of an expression. Neither is there a mechanism for initializing a variable without explicitly stating its type. Stroustrup suggests [Str02] the language to be extended with mechanisms for both these tasks, discussing broadly several different possibilities for the syntax and semantics of these mechanisms.

The emphasis of this proposal is on defining the exact semantics for the above two mechanisms. We do, however, suggest a specific syntax: the ***decltype*** operator for querying the type of an expression, and the keyword ***auto*** for indicating that the compiler should deduce the type of a variable from its initializer expression. Moreover, we propose a variant of one of the new function definition syntaxes discussed in [Str02].

In the following, we summarize earlier discussions on ***typeof***. We use the operator name ***typeof*** when referring to the mechanism for querying a type of an expression in general. The ***decltype*** operator refers to the proposed variant of ***typeof***.

1.1 Motivation

C++ would benefit from ***typeof*** and ***auto*** in many ways. These features would be convenient on several occasions, and increase the readability of the code. More importantly, the lack of a ***typeof*** operator is worse than an inconvenience for many generic library authors: it is often not possible to express the return type of a generic function. This leads to hacks, workarounds, and reduced functionality with an additional burden imposed on the library user (see for example the return type deduction mechanisms in [JPL03, Dim01, WK02, Vel], or the function object classes in the standard library). Below we describe typical cases which would benefit from ***typeof*** or ***auto***. For additional examples, see [Str02].

- The return type of a function template can depend on the types of the arguments. It is currently not possible to express such return types in all cases. Many forwarding functions suffer from this problem. For example, what should be the return type of the ***trace*** function below?

```
template <class Func, class T>
??? trace(Func f, T t) { std::cout << "Calling f"; return f(t); }
```

Currently, return types that depend on the function argument types are expressed as (complicated) metafunctions that define the mapping from argument types to the return type. For example:

```
template <class Func, class T> typename Func::result_type trace(Func f, T t);
```

or following a recent library proposal [Gre03]:

```
template <class Func, class T>
typename result_of<Func(T)>::type trace(Func f, T t);
```

Such mappings rely on programming conventions and can give incorrect results. It is not possible to define a set of traits classes/metafunctions that cover all cases. With ***typeof*** (that has appropriate semantics, see Section 2) the ***trace*** function could be defined as:

```
template <class Func, class T>
fun trace(Func f, T t) −> typeof(f(t));
```

Note the suggested new function definition syntax, discussed in Section 5, where the return type expression following the −> symbol comes after the argument list. Using this syntax, the argument names are in scope in the return type expression.

As another example, the return types of operators in various algebraic libraries (computations on vectors, matrices, physical units, etc.) commonly depend on the argument types in non-trivial ways. We show an addition operator between two matrices as an example:

```
template <class T> class matrix;
  ...
template <class T, class U>
??? operator+(const matrix<T>& t, const matrix<U>& u);
```

For instance, suppose the return type of ***matrix<int>() + matrix<double>()*** is ***matrix<double>***. Expressing such relations requires heavy template machinery. Using ***typeof***, the relation could be expressed as:

```
template <class T, class U>
fun operator+(matrix<T> t, matrix<U> u) −> matrix<typeof(t(0,0)+u(0,0))>;
```

- Often the type of a relatively simple expression can be very complex. It can be tedious to explicitly write such types, making it tedious to declare variables. Common cases are iterator types of containers:

```
int foo(const std::map<std::string, std::map<std::string, std::string>& m) {
  std::map<std::string, std::map<std::string, std::string> >::const_iterator
    it = m.begin();
  ...
}
```

Types resulting from invocations of function templates can be too complicated to be at all practical to write by hand. For example, the type of the Lambda Library [JP02] expression ***_1 + _2 + _3*** spans several lines, and contains types that are not part of the public interface of the library. Some variant of ***typeof*** can be used to address this problem. For example, the declaration of ***it*** using ***typeof*** becomes:

```
  typeof(m.begin()) it = m.begin();
```

This is obviously a great improvement. However, as described in Section 4, the semantics for ***typeof*** is not exactly ideal for the purpose of declaring variables. Furthermore, the redundant repetition of the initializer expression is a distraction and not quite harmless. For example, the following example appeared (innocently) in a reflector discussion:

```
  typeof(x∗y) z = y∗x;
```

The snag is that the type of similar, yet different, expressions are not necessarily the same. Thus, the need to repeat the initializer becomes a maintenance problem. Consequently, we propose a separate mechanism for declaring variables, ***auto***, which deduces the type of the variable from its initializer expression:

```
auto it = m.begin();
```

2 Design alternatives for *typeof*

Two main options for the semantics of a ***typeof*** operator have been discussed: either to preserve or to drop references in types. For example:

```
int& foo();
...
typeof(foo());   // int& or int?

int a;
int& b = a;

typeof(a);       // int& or int?
typeof(b);       // int& or int?
```

A reference-dropping ***typeof*** always removes the top-level references. Some compiler vendors (EDG, Metrowerks, GCC) provide a ***typeof*** operator as an extension with reference-dropping semantics. As described in Section 4, this appears to be ideal for expressing the type of variables. On the other hand, the reference-dropping semantics fails to provide a mechanism for exactly expressing the return types of generic functions, as demonstrated by Stroustrup [Str02]. This implies that a reference-dropping ***typeof*** would cause problems for writers of generic libraries. A reference-preserving ***typeof*** has been proposed to return a reference type if its expression operand is an *lvalue*. However, such semantics could easily confuse programmers and lead to surprises. For example, in the above example ***a*** is declared to be of type ***int***, but under a ***typeof*** reflecting "lvalueness", ***typeof(a)*** would be ***int&***. It seems that variants of both semantics are required, thus suggesting the need for two different ***typeof***-like operators. We believe, however, that just one ***typeof*** operator is enough.

In the standard text, ‘type of an expression’ refers to the non-reference type, Section 5(6)[1]:

> If an expression initially has the type “reference to ***T***” (8.3.2, 8.5.3), the type is adjusted to ***T*** prior to any further analysis, the expression designates the object or function denoted by the reference, and the expression is an lvalue.

For example:

```
int x;
int xx = x;        // type of the expression x is int
int& y = x;
int yy = y;        // type of the expression y is int
int& foo();
int zz = foo();    // type of the expression foo() is int
```

The lvalueness of an object is expressed separate from its type. However, in the program text, a reference is clearly part of the type of an expression. From here on, we refer to the type in the program text as the *declared type* of an object.

```
int x;          // declared type of x is int
int& y = x;     // declared type of y is int&
int& foo();     // declared type of foo() is int& (because the declared return type of foo is int&)
```

The first line above demonstrates that lvalueness of an object does not imply that the declared type of the object is a reference type. In this proposal, the semantics of the operator that provides information of the type of expressions reflects the declared type. Therefore, we propose the operator to be named ***decltype***.

[1]The standard is not always consistent in this respect, in some occasions reference is part of the type.

3 The *decltype* operator

The syntax of ***decltype*** is:

```
unary−expression
   ...
   decltype ( unary−expression )
   decltype ( type−id )
   ...
```

We require parentheses (as opposed to ***sizeof***'s more liberal rule) to keep the syntax simple and to keep the door open for inquiry operations on the results of ***decltype***, e.g. ***decltype(T).is_reference()***. However, we do not propose any such extensions. The semantics of the ***decltype*** operator are described as:

1. If ***e*** is a name of a variable in namespace or local scope, a static member variable, or a formal parameter of a function, ***decltype(e)*** is the declared type for that variable or formal parameter. Particularly, ***decltype(e)*** results in a reference type only if the variable or formal parameter is declared as a reference type.
2. If ***e*** is an invocation of a function or operator, either user-defined or built-in, ***decltype(e)*** is the declared return type of that function. The standard text does not list the prototypes of all built-in operators. For the operators and expressions whose prototypes are not listed, the declared type is a reference type whenever the return type of the operator is specified to be an lvalue.
3. ***decltype*** does not evaluate its argument expression.
4. The ***decltype*** taking a type parameter is an identity function: ***decltype(T)*** is equal to ***T*** for any type expression ***T***.

The last rule is for consistency with the ***sizeof*** operator. However, it would have the effect of allowing a type expression containing commas to be passed into a macro as a single macro argument, without enclosing the expression into parenthesis (which are not allowed around type expressions in many context). For example:

```
SOMEMACRO(pair<int, int>)            // two arguments
SOMEMACRO(decltype(pair<int, int>))  // one argument
```

In the following we give examples of ***decltype*** with different kinds of expressions:

- Function invocations:

```
int foo();
decltype(foo())      // int

float& bar(int);
decltype (bar(1))   // float&

decltype(1+2)       // int

int i;
decltype (i = 5)    // int&, because the "declared type" of integer assignment is int&

class A { ... };
const A bar();
decltype (bar())    // const A

const A& bar2();
decltype (bar2())    // const A&
```

- Variables in namespace or local scope:

```
int a;
int& b = a;
const int& c = a;
const int d = 5;
const A e;

decltype(a)   // int
decltype(b)   // int&
decltype(c)   // const int&
decltype(d)   // const int
decltype(e)   // const A
```

- Formal parameters of functions:

```
void foo(int a, int& b, const int& c, int* d) {
  decltype(a)   // int
  decltype(b)   // int&
  decltype(c)   // const int&
  decltype(d)   // int*
  ...
}
```

- Function types:

```
int foo(char);
decltype(foo)     // int(char)
decltype(&foo)   // int(*)(char)
```

  Note that objects of function types cannot exist:

```
decltype(foo) f1 = foo;     // error, we can't have a variable of type int(char)
decltype(foo)* f2 = foo;   // fine: f2 is an int(*)(char)
decltype(foo)& f3 = foo;   // fine: f3 is an int(&)(char)
```

- Array types:

```
int a[10];
decltype(a);   // int[10]
```

- Pointers to member variables and member functions:

```
class A {
  ...
  int x;
  int& y;
  int foo(char);
  int& bar() const;
};

decltype(&A::x)      // int A::*
decltype(&A::y)      // error: pointers to reference members are disallowed (8.3.3 (3))
decltype(&A::foo)   // int (A::*) (char)
decltype(&A::bar)   // int& (A::*) (char) const
```

- Member variables:

  Member variables are handled according to clause 5.1 (7) in the standard:

  > Within the definition of a nonstatic member function, an identifier that names a nonstatic member is transformed to a class member access expression (9.3.1).

The type given by ***decltype*** is thus the return type specified for member access expressions, and is a reference type whenever the return type is an lvalue. Also, const and volatile qualifiers of the member function will be added to the member variable type as described in Section 5.2.5 in the standard. Static member variables are treated as variables in namespace scope.

```
class A {
  int a;
  int& b;
  static int c;

  void foo() {
    decltype(a);   // int&
    decltype(b);   // int&
    decltype(c);   // int
  }

  void bar() const {
    decltype(a);   // const int&
    decltype(b);   // int&
    decltype(c);   // int
  }
 ...
};

A an_A;
decltype(an_A.a)   // int&
decltype(A().a)      // int
```

The expression ***an_A*** is an lvalue, and thus, according to 5.2.5(4), the expression ***an_A.a*** is also an lvalue. Consequently, rule 2 above implies that the declared type is ***int&***. The expression ***A()***, on the other hand, is an rvalue, so ***A().a*** is an rvalue as well. The declared type of this expression is thus ***int***.

Whether member variable names used outside of member function bodies should be considered to be member access expressions or not is not an issue; member variable names are not in scope in the class declaration scope:

```
class B {
  int a;
  enum { b };

  decltype(a) c;                          // error, a not in scope
  static const int x = sizeof(a);   // error, a not in scope

  decltype(this−>a) c2;           // error, this not in scope
  decltype(((B∗)0)−>a) hack;  // ok

  decltype(a) foo() { ... };               // error, a not in scope
  fun bar() −> decltype(a) { ... };   // still an error

  decltype(b) enums_are_in_scope() { return b; } // ok
  ...
};
```

Should this be seen as a serious restriction, we can consider relaxing it, but we see no current need for that.

- ***this***:

```
class X {
  void foo() {
    decltype(this)     // X∗
    decltype(∗this)   // X&
```

```
        ...
        }
        void bar() const {
          decltype(this)     // const X*
          decltype(*this)   // const X&
        ...
        }
      };
```

- Literals:

  5.1(2) states that string literals are lvalues, all other literals rvalues. Consequently, the ***decltype*** of a string literal is a references, and all other literals are non-reference types:

  ```
  decltype("decltype")   // const char(&)[9]
  decltype(1)                 // int
  ```

Catering to library authors The semantics of ***decltype*** described above allow to return types of forwarding functions to be accurately expressed in all cases. The ***trace*** and matrix addition examples in Section 1 work as expected with this definition of ***decltype***.

Catering to novice users The rules are consistent; if ***expr*** in ***decltype(expr)*** is a variable or formal parameter the programmer can trace down the variable's or parameter's declaration, and the result of ***decltype*** is exactly the declared type. If ***expr*** is a function invocation, the programmer can perform manual overload resolution; the result of the ***decltype*** is the return type in the prototype of the best matching function. The prototypes of the built-in operators are defined by the standard, and if some are missing, the rule that an lvalue has a reference type applies.

4 Auto

Stroustrup brought up the idea of reviving the ***auto*** keyword to indicate that the type of a variable is to be deduced from its initializer expression [Str02]. For example:

```
auto x = 1; // x has type int
```

auto is faced with the same questions as the mechanism for querying the type of an expression. Should references be preserved or dropped? Should ***auto*** be defined in terms of ***decltype*** (i.e., is ***auto var = expr*** equivalent to ***decltype(expr) var = expr***)? We suggest that the answer to that question be "no" because the semantics would be surprising, non-ideal for the purpose of initializing variables, and incompatible with current uses of ***typeof***. Instead, we propose that the semantics of ***auto*** follow exactly the rules of template argument deduction. The ***auto*** keyword can occur in any deduced context in an expression. Examples (the notation ***x : T*** is read as "***x*** has type ***T***"):

```
int foo();
auto x1 = foo();              // x1 : int
const auto& x2 = foo();   // x2 : const int&
auto& x3 = foo();           // x3 : int&: error, cannot bind a reference to a temporary

float& bar();
auto y1 = bar();              // y1 : float
const auto& y2 = bar();   // y2 : const float&
auto& y3 = bar();           // y3 : float&
```

A major concern in discussions of ***auto*** like features has been the potential difficulty in figuring out whether the declared variable will be of a reference type or not. Particularly, is unintentional aliasing or slicing of objects likely? For example

```
// ...
auto b = d;   // is this casting a reference to a base or slicing an object?
b.f();           // is polymorphic behavior preserved?
auto x = y;   // is this value semantics (copying) or reference semantics?
```

A unconditionally reference-preserving ***auto*** (e.g. an ***auto*** directly based on ***decltype***) would favor an object-oriented style of use to the detriment of types with value semantics. Basing ***auto*** on template argument deduction rules provides a natural way for a programmer to express his intention. Controlling copying and referencing is essentially the same as with variables whose types are declared explicitly. For example:

```
A foo();
A& bar();
...
A x1 = foo();       // x1 : A
auto x1 = foo();   // x1 : A

A& x2 = foo();      // error, we cannot bind a non−lvalue to a non−const reference
auto& x2 = foo();  // error

A y1 = bar();       // y1 : A
auto y1 = bar();   // y1 : A

A& y2 = bar();      // y2 : A&
auto& y2 = bar();  // y2 : A&
```

Thus, as in the rest of the language, value semantics is the default, and reference semantics is provided through consistent use of ***&***. The type deduction rules extend naturally to more complex definitions:

```
std::vector<auto> x = foo();
std::pair<auto, auto>& y = bar();
```

The declaration of ***x*** would fail at compile time if the return type of foo was not an instance of ***std::vector***. Analogously, the return type of ***bar*** must be an instance of ***std::pair***. Declaring such partial types for variables can be seen as documenting the intent of the programmer. Here, the compiler can enforce that the intent is satisfied.

Stroustrup [Str02] points out that an ***auto*** like facility is primarily for declaring local variables, but unless deliberately restricted, can be used in other contexts as well:

```
template <class T> void f(T a, auto b = a+2);
```

We do not yet have a clear picture, whether it would be beneficial, harmful, or neither, to allow such uses of ***auto***. Thus, the current proposal restricts its use to variable declarations. The rules can be relaxed later.

4.1 Implicit templates

By defining the semantics of ***auto*** in terms of initialization, we automatically open the door for allowing ***auto*** in every context where a type is deduced through the initialization rules. Using ***auto*** as a mechanism for *implicit template functions* was presented in [Str02] and has been discussed within the Evolution Working Group. We do not propose to allow implicit template functions, at least not as the first step. Nevertheless, the proposed semantics of ***auto*** provides a consistent basis for implicit templates: every occurrence of ***auto*** can be regarded as a new unique template parameter. For example:

```
void foo(auto a, auto& b, const auto& c, pair<int, auto> d, auto* e);
```

can be defined as being equivalent to:

```
template<class __A, class __B, class __C, class__D, class __E>
void foo(__A a, __B& b, const __C& c, pair<int, __D> d, __E* e);
```

Going further in this direction, ***auto*** could be used as the return type of a function:

```
auto add(auto x, auto y) { return x + y; }
```

The return type would be deduced as the type ***ret*** in the expression: ***auto ret = x + y***. Any deduced context would be allowed:

```
const auto* foo(...);
auto& bar(...);
vector<auto> bah(...);
```

Again, we do not currently propose allowing the use of ***auto*** in the return type. We mention it to demonstrate the generality of the proposed mechanism and semantics. For completeness, we document some of the issues concerning ***auto*** return types:

- Multiple ***return*** statements are a problem. The two solutions are either not allowing ***auto*** in the return type of a function with more than one return statement, or applying type deduction rules similar to ones used for deducing the type of the conditional operator invocation. The former solution seems more plausible.
- Missing return statement. Should the return type be ***void***, or should such a function definition be an error?
- To be able to deduce the return type from the body of the function, the body needs to be accessible, which would restrict a function with an ***auto*** return type to be callable only from the compilation unit that contains the definition of the function.

5 New function declaration syntax

We can anticipate that a common use for the ***decltype*** operator is to specify return types that depend on the types of function arguments. Unless their argument names are in scope in the return type expression, this task becomes unnecessarily complicated. For example:

```
template <class T, class U> decltype((*(T*)0)+(*(U*)0)) add(T t, U u);
```

The expression ***(∗(T∗)0)*** is a hackish way to write an expression that has the type ***T*** and doesn't require ***T*** to be default constructible. If the argument names were in scope, the above declaration could be written as:

```
template <class T, class U> decltype(t+u) add(T t, U u);
```

To allow the argument names to be in scope in the return type expression, several syntaxes that move the return type expression after the argument list are discussed in [Str02]. If the return type expression comes before the argument list, parsing becomes difficult and name lookup may be less intuitive; the argument names may have other uses in an outer scope at the site of the function declaration.

It is not strictly necessary to have the argument names in scope: Every return type expression that can be written using the argument names can also be written using only the types of the arguments — at the cost of verbosity and loss of readability.

From the syntaxes proposed in [Str02], and discussed within the evolution group in the Oxford-03 meeting, we suggest adding a new keyword ***fun*** to express that the return type is to follow after the argument list. The return type expression is preceded by −> symbol, and comes after the argument list (and potential cv-qualifiers in member functions) but before the exception specification:

```
template <class T, class U> fun add(T t, U u) −> decltype(t + u);
class A {
  fun f() const −> int throw ();
};
```

We refer to [Str02] for further analysis on the effects of the new function declaration syntax[2] and expect a more detailed paper addressing issues such as pointers to functions to be written.

6 Conclusions

In C++2003, it is not possible to express the return type of a function template in all cases. Furthermore, expressions involving calls to function templates commonly have very complicated types, which are practically impossible to write by hand. Hence, it is often not feasible to declare variables for storing the results of such expressions. This proposal describes ***decltype*** and ***auto***, two closely related language extensions that solve these problems. Intuitively, the ***decltype*** operator returns the declared type of an expression. For variables and parameters, this is the type the programmer finds

[2]Adding a new keyword is a drastic measure. Looking at the future, however, ***fun*** can serve as part of the syntax for defining unnamed functions. No formal proposal of such extension exists at the moment, though.

in the program text. For functions, the declared type is the return type of the definition of the outermost function called within the expression, which can also be traced down and read from the program text (or in the standard in the case of built-in functions). The semantics of ***auto*** is unified with template argument deduction, which gives a natural means to declare variables whose types are deduced from their initializer expressions.

We recognize the order of importance of the features discussed in this proposal:

1. ***decltype***. The lack of ***decltype*** is a source of frustration in generic programming, and has lead to extremely complicated and brittle library solutions, which can only (poorly) approximate a built-in ***decltype*** operator.

2. ***auto***. Given ***decltype***, ***auto*** can be (poorly) emulated. However, because of the subtlety of such emulation and its difference from current practice in expressing declarations, we regard ***auto*** as important as ***decltype***.

3. New function declaration syntax. It is convenient to be able to use function parameters within a ***decltype*** operator in return type expression. This necessitates new function declaration syntax, where the return type comes after the parameter list. The new syntax is not strictly necessary; again, at the expense of increased complexity and verbosity, all return types can be expressed without the new syntax.

4. Implicit templates using ***auto***. Can be safely left to be considered again in the future. We do not see reasons why this extensions would be particularly difficult to implement.

5. Allowing the use of ***auto*** in the return type expression. Can be safely left to maybe be considered again in the future.

Finally, the main goal of the proposal is to define the semantics of ***decltype*** and ***auto***, rather than completely discuss the syntax. However, we do consider the proposed syntax, the best (or at least the least bad) among alternatives.

7 Acknowledgements

We are grateful to Gary Powell, Mat Marcus, Daveed Vandevoorde, David Abrahams, Andreas Hommel, Jeremiah Willcock, and Paul Mensonides for their valuable input in preparing this proposal. Clearly, this proposal builds on input from members of the EWG as expressed in face-to-face meetings and reflector messages.

References

[Dim01] Peter Dimov. *The Boost Bind Library*. Boost, 2001. `http://www.boost.org/libs/bind`.

[Gre03] Douglas Gregor. A uniform method for computing function object return types. C++ standards committee document N1437=03-0019, February 2003.

[JP02] Jaakko Järvi and Gary Powell. *The Boost Lambda Library*, 2002. `http://www.boost.org/libs/lambda`.

[JPL03] Jaakko Järvi, Gary Powell, and Andrew Lumsdaine. The Lambda Library : unnamed functions in C++. *Software—Practice and Experience*, 33:259–291, 2003.

[Str02] Bjarne Stroustrup. Draft proposal for "typeof". C++ reflector message c++std-ext-5364, October 2002.

[Vel] Todd Veldhuizen. Blitz++ home page. `http://oonumerics.org/blitz`.

[WK02] Jörg Walter and Mathias Koch. *uBLAS Library*. Boost, 2002. `http://www.boost.org/libs/numeric`.